آزمایش چهار

خلاصه ی آزمایش:

در این آزمایش، دانشجویان با فرایند کامپایل آشنا شده و مروری بر کد C و assembly انجام خواهد شد.

اهداف آزمایش:

- آشنایی با GCC
- مرور کد C و آشنایی با GDB
- آشنایی با اسمبلی ARM
- راهنمایی در مورد نحوه ی پیاده سازی stack به زبان C

تجهیزات مورد نیاز:

- برد Raspberry pi II و کابل برق آن
- کارت Micro SD
- کامپیوتر (به همراه موس و کیبورد)
- کابل شبکه جهت ارتباط برد با کامپیوتر
- نرم افزار Putty جهت راه اندازی SSH

شرح آزمایش:

✓ بخش "کامپایل و اجرای برنامه ی C" را از فصل " مرور کد C و آشنایی با GDB " اجرا کنید.

✓ فایلی با نام test01.c در `~/session04/c` ایجاد کنید.

- ✓ برنامه ای به زبان C بنویسید که 20 کارکتر بین a تا z را به صورت تصادفی به کاربر نمایش دهد و پس از اینکه کاربر کل کارکتر ها را تایپ کرد، سرعت و دقت تایپ را محاسبه کند و نمایش دهد. کد را کامپایل کرده و اجرا و تست کنید.
- ✓ بخش "اسمبل و اجرای برنامه ی assembly " از قسمت " آشنایی با اسمبلی ARM" را اجرا کنید.
- ✓ فایل هایی با نام mul01.s و mul02.s وmul03.s در `~/session04/asm/` ایجاد کنید.
- ✓ به سه طریق برنامه ای به زبان اسمبلی بنویسید که عدد 5 را در عدد 8 ضرب کند و نتیجه را در تابع main ، برگرداند.
- ✓ فایل با نام compare01.s در `~/session04/asm/` ایجاد کنید.
- ✓ برنامه ای به زبان اسمبلی بنویسید که دو خانه از حافظه با مقدار اولیه را بخواند و اگر عدد اول بزرگتر از عدد دوم بود، 1 ، درغیر این صورت 2 را در تابع main ، برگرداند.
- ✓ فایل هایی با نام mul01.c و compare01.c در `~/session04/asm/` ایجاد کنید.
- ✓ برنامه هایی به زبان C بنویسید که کاری که برنامه هایی mul01.s وcompare01.s انجام می دهند را انجام دهند.آنها را کامپایل و اجرا کنید. توسط GDB فایل های اجرایی را disassemble کنید و با فایل های اسمبلی خود مقایسه کنید.
- ✓ برنامه ی stack را با پیاده سازی به روش linked list به زبان C بنویسید.

وظایف:

1- بخش مطالعه ی این دستور کار به طور کامل مطالعه شود.

منابع مرتبط:

مطالعه:

آشنایی با GCC

مرور کد C و آشنایی با GDB

آشنایی با اسمبلی ARM

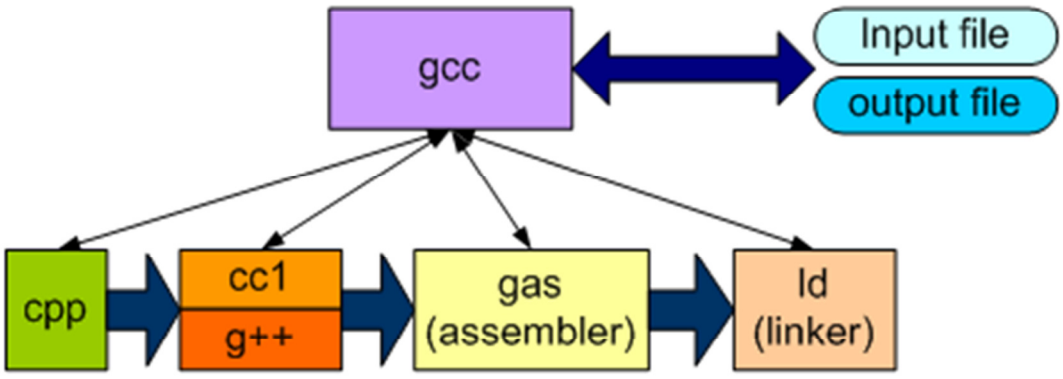
راهنمایی در مورد نحوه ی پیاده سازی stack به زبان C

آشنایی با GCC

GNU Binutils

- The GNU binutils are a collection of binary tools
  - **ld** - the GNU linker
  - **as** - the GNU assembler
  - Other binary tools
    - ar - A utility for creating, modifying and extracting from archives
    - gprof - Displays profiling information
    - **objcopy** - Copys and translates object files
    - objdump - Displays information from object files
    - ...
- Easy to port **binutils** to other platforms
- Suitable for embedded-system development

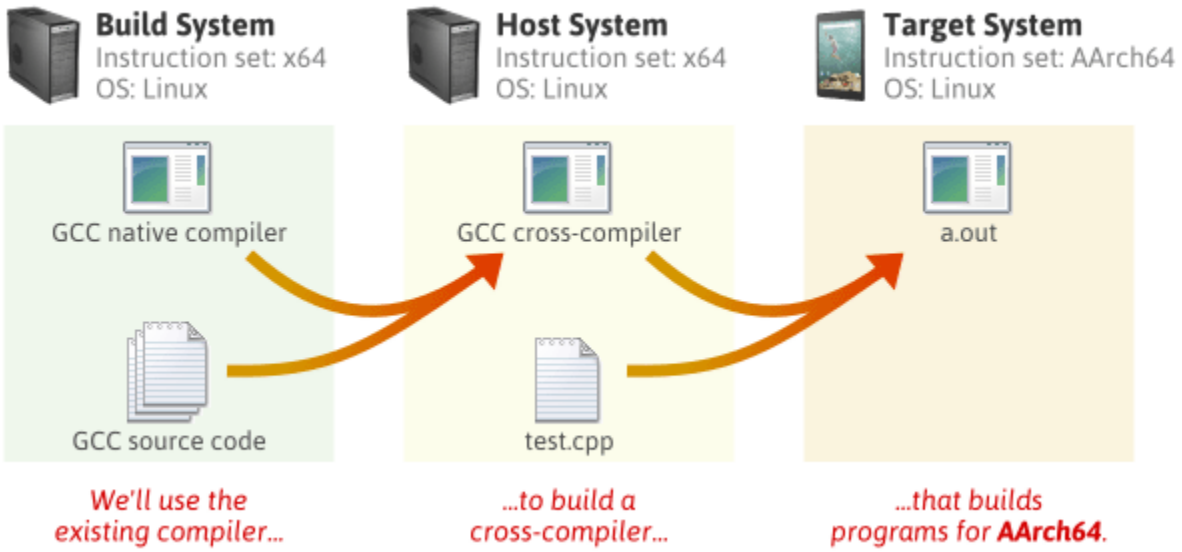
GCC Execution



توضیح Cross-Compiler

زمانی که یک برنامه به هر زبان برنامه نویسی مانند C نوشته شود، برای اجرا شدن باید توسط یک کامپایلر به زبان ماشین تبدیل شود که به این عمل کامپایل کردن گویند. هر خانواده از پردازنده ها دارای مجموعه دستورالعمل هایی هستند که اصطلاحاً زبان ماشین نامیده می شوند. بنابراین یک کامپایلر برای تولید زبان ماشین باید با معماری پردازنده مانند ثبات ها، دستورالعمل ها و ... آشنایی داشته باشد.

در حالت کلی زمانی که یک برنامه را در یک سیستم کامپایل می کنیم، پردازنده ی آن سیستم به عنوان مقصد کامپایل در نظر گرفته شده و زبان آن به عنوان زبان ماشین درنظر گرفته می شود. در مواردی نیاز است که علیرغم اجرای سیستم عامل بر روی یک پردازنده، کامپایلر زبان ماشین دیگری را به عنوان مقصد انتخاب نماید. در چنین مواردی کامپایلر را اصطلاحا Cross-Compiler گویند. برای مثال اکثر سیستم های خانگی از پردازنده های خانواده Intel استفاده می کنند. بنابراین، کامپایلرها در زمان کامپایل یک برنامه، آن را با توجه به دستورالعمل های پردازنده های خانواده Intel کامپایل می کنند. حال اگر بخواهیم در یک سیستم خانگی، یک برنامه را برای اجرا در پردازنده های ARM کامپایل نماییم، از یک Cross-Compiler استفاده می کنیم تا زبان مقصد را بر اساس دستورالعمل های پردازنده ARM در نظر بگیرد. از Cross-Compiler های معروف می توان به Eclipse اشاره نمود.



برخی از دلایلی که به جای کامپایل کردن برنامه روی Target platform ، از Cross-Compiler استفاده می کنیم این است که:

- سرعت:

معمولا سرعت Target platform پایین تر از سرعت کامپیوتری است که به عنوان Cross-Compiler می خواهیم از آن استفاده کنیم.

- قابلیت:

کامپایل کردن به منابع زیادی احتیاج دارد و Target platform معمولا رم خیلی زیاد و یا هارد زیاد ندارد. در برخی موارد حتی امکان اجرا کردن کامپایلر در سیستم هدف وجود ندارد.

- راحتی:

در Target platform ممکن است که صفحه ی نمایش و CD-ROM و صفحه کلید نداشته باشد و نیز در سیستم میزبان ما می توانیم نرم افزار های کمکی داشته باشیم به عنوان مثال editor های بهتر.

مرور کد C و آشنایی با GDB

در این بخش ابتدا یک کد c را روی برد خود اجرا خواهید کرد و در انتها برخی موارد در مورد زبان c یادآوری می شود.

کامپایل و اجرای برنامه ی C :

ابتدا باید کد خود را در سیستم ذخیره کنید. برای این کار ابتدا فولدر مناسب را ایجاد کنید. این کار را توسط دستور mkdir انجام می دهید.

MKDIR

Makes a new directory, e.g. `mkdir newDir` would create the directory `newDir` in the present working directory.

```
mkdir ~/session04/
```

```
mkdir ~/session04/C
```

سپس باید درون فولدر ایجاد شده بروید. برای این کار از دستور CD استفاده می کنیم.

CD

Changes the current directory to the one specified. Can use relative (i.e. `cd directoryA`) or absolute (i.e. `cd /home/pi/directoryA`) paths.

```
pi@raspberrypi ~ $ cd ~/session04/c/
pi@raspberrypi ~/session04/c $
```

سپس باید فایل با نام مناسب ایجاد کنید.برای ایجاد فایل از دستور touch استفاده می کنیم.

TOUCH

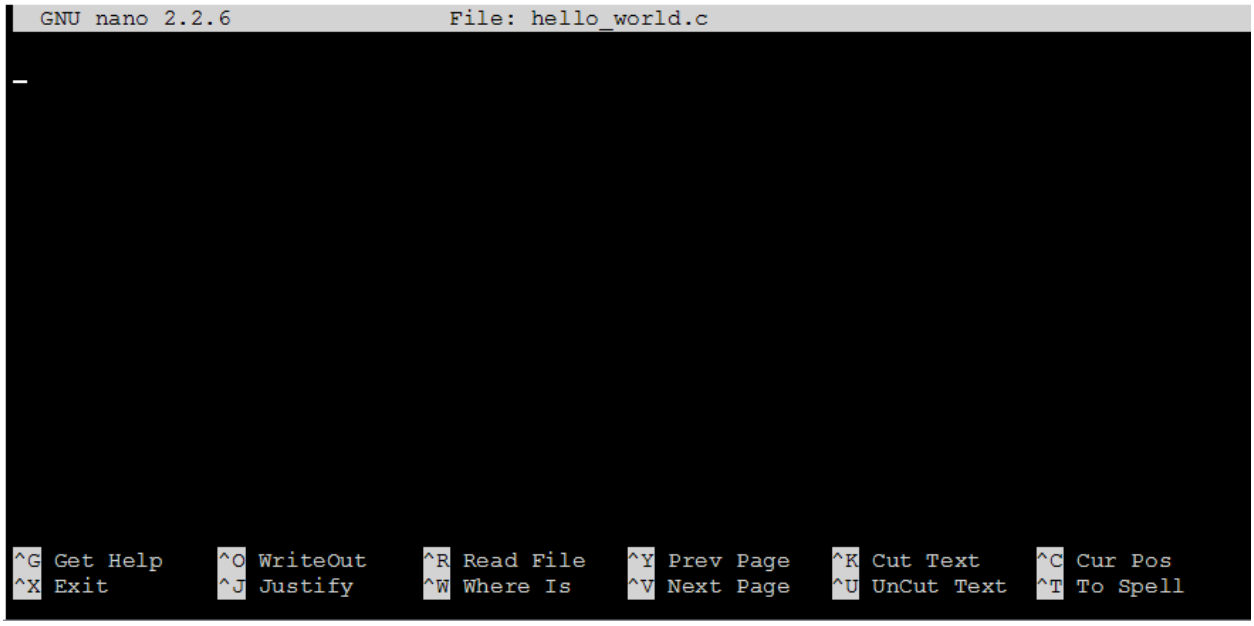
Either sets the last modified time-stamp of the specified file(s) or creates it if it does not already exist.

```
pi@raspberrypi ~/session04/c $ touch hello_world.c
```

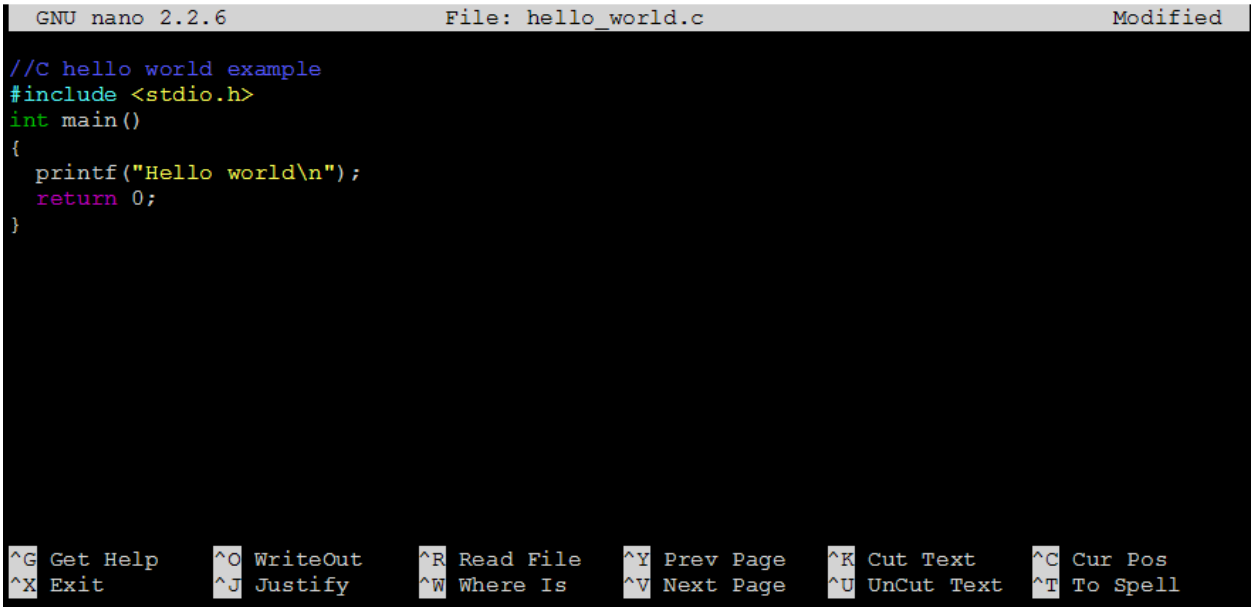
برای نوشتن کد به یک text editor احتیاج دارید. یکی از ساده ترین text editor ها که می توانید استفاده کنید، nano است.

```
pi@raspberrypi ~/session04/c $ nano hello_world.c
```

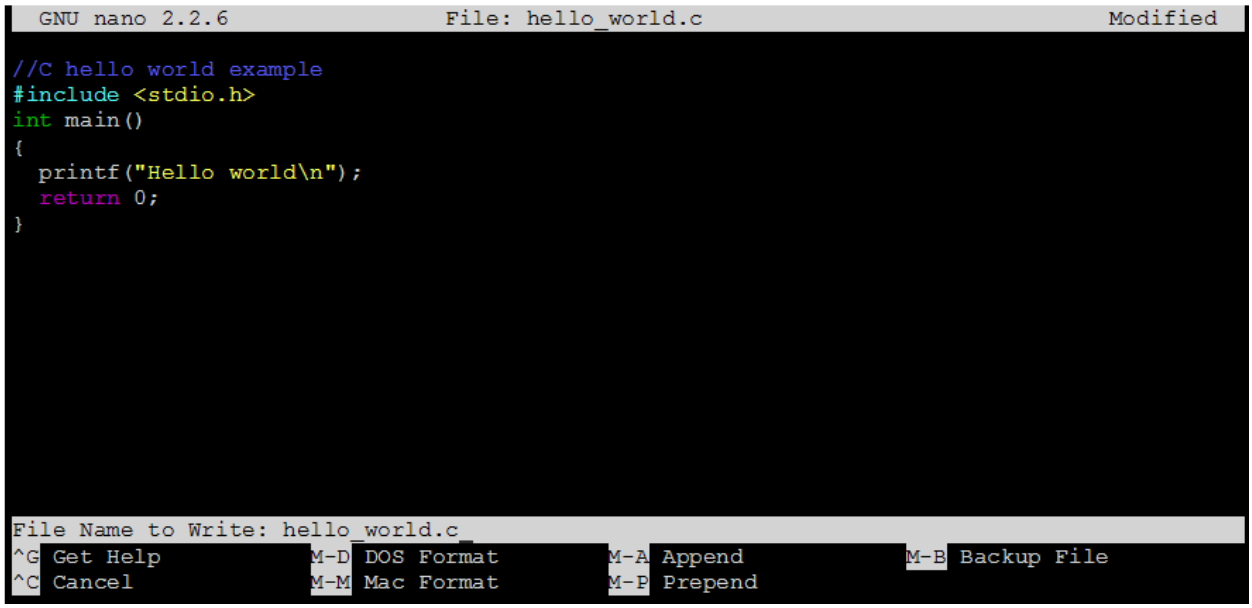
در محیط nano همانطور که در شکل زیر مشاهده می کنید، می توانید متن اضافه کنید و کارهای کنترلی که می توانید انجام دهید نیز در پایین صفحه به شما نمایش داده شده است.



حال باید برنامه را تایپ کنید .



برای save کردن برنامه باید از ctrl+o استفاده کنید. پس از اینکه این کلید ها را زدید، همانطور که در شکل زیر مشاهده می کنید، از شما نام فایل که قرار است ذخیره شود پرسیده می شود. با فشردن enter نام قبلی را برای فایل تایید کنید.



برای خروج از nano باید از ctrl+x استفاده کنید.

سپس باید فایل را کامپایل کنید تا به فایل اجرایی تبدیل شود. برای این کار از gcc استفاده می کنید.

```
pi@raspberrypi ~/session04/c $ gcc -o hello_world hello_world.c
```

پس از آن فایل اجرایی ایجاد شده را اجرا می کنید.

```
pi@raspberrypi ~/session04/c $ ./hello_world
Hello world
pi@raspberrypi ~/session04/c $
```

آشنایی با GDB:

GDB debugger

The Unix/Linux Debugger: gdb

When all else fails…

- Stop the program
- Look at (or modify) registers
- Look at (or modify) memory
- Single-step the program
- Set a “breakpoint”

To compile a program for use with gdb

… use the ‘-g’ compiler switch

Controlling program execution

run

Start the program.

step

Step program until it reaches a different source line.

next

Step program, proceeding through subroutine calls.

Single step to the next source line, not into the call.

Execute the whole routine at once; stop upon RETURN.

continue

Continue program execution after signal or breakpoint.

Controlling program execution

break, del

Set and delete breakpoints at particular lines of code

watch, rwatch, awatch

Data breakpoints

Stop when the value of an expression changes (watch), when expression is read (rwatch), or either (awatch)

Printing out code and data

print

print expr

```
(gdb) print x
(gdb) print argv[0]
```

print {type} addr

```
(gdb) p {char *} 0xbfffdce4
```

```
(gdb) print/x addr
```

'/x' says to print in hex. See "help x" for more formats

Same as examine memory address command (x)

printf "format string" arg-list

```
(gdb) printf "%s\n", argv[0]
```

list

Display source code

Other Useful Commands

where, backtrace

Produces a backtrace (the chain of function calls that brought the program to its current place).

up, down

Change scope in stack

info

`(gdb) info`	prints a list of info commands
`(gdb) info br`	Print a table of all breakpoints and watchpoints
`(gdb) info r`	The machine registers

quit

Exit gdb

Different gdb interfaces

Better graphical interfaces

Most debuggers provide the same functionality

- `gdb –tui`
  Sort of graphical (like “vi”)
- Insight: http://sourceware.org/insight
- DDD: http://www.gnu.org/software/ddd
- TDB: http://pdqi.com/browsex/TDB.html
- KDbg: http://www.kdbg.org

مروری بر برنامه نویسی C :

The C Standard Library

Common functions we don't need to write ourselves

A portable interface to many system calls

Analogous to class libraries in Java or C++

Function prototypes declared in standard header files

Must include the appropriate ".h" in source code

```
#include <stdio.h>          #include <stddef.h>
#include <time.h>           #include <math.h>
#include <string.h>         #include <stdarg.h>
#include <stdlib.h>
```

"man 3 printf" shows which header file to include

K&R Appendix B describes the functions

Code linked in automatically

At compile time (if statically linked, `gcc -static`)

At run time (if dynamically linked)

Use "ldd" command to list dependencies

The C Standard Library

I/O stdio.h

`printf, scanf, puts, gets, open, close, read, write,`
`fprintf, fscanf, fseek, …`

Memory and string operations string.h

`memcpy, memcmp, memset,`
`strlen, strncpy, strncat, strncmp,`
`strtod, strtol, strtoul, …`

Character Testing ctype.h

`isalpha, isdigit, isupper,`
`tolower, toupper, …`

Argument Processing stdarg.h

`va_list, va_start, va_arg, va_end, …`

The C Standard Library

Utility functions stdlib.h

`rand, srand, exit, system, getenv,`
`malloc, free, atoi, …`

Time time.h

`clock, time, gettimeofday, …`

Jumps setjmp.h

`setjmp, longjmp, …`

Processes unistd.h

`fork, execve, …`

Signals signals.h

`signal, raise, wait, waitpid, …`

Implementation-defined constants limits.h, float.h

`INT_MAX, INT_MIN, DBL_MAX, DBL_MIN, …`

Formatted Output

```
int printf(char *format, …)
```
Sends output to standard output

```
int fprintf(FILE *stream, char *format, …);
```
Sends output to a file

```
int sprintf(char *str, char *format, …)
```
Sends output to a string variable

Return Value: The number of characters printed
(not including trailing \0)

On Error: A negative value is returned.

Formatted Output

The format string is copied as-is to output.

Except the % character signals a formatting action.

Format directives specifications

- Character (%c), String (%s), Integer (%d), Float (%f)
- Fetches the next argument to get the value
- Formatting commands for padding or truncating output and for left/right justification
  - %10s → Pad short string to 10 characters, right justified
  - %-10s → Pad short string to 10 characters, left justified
  - %.10s → Truncate long strings after 10 characters
  - %10.15s → Pad to 10, but truncate after 15, right justified

For more details: `man 3 printf`

Formatted Input

```
int scanf(char *format, …)
```
Read formatted input from standard input

```
int fscanf(FILE *stream, const char *format, ...);
```
Read formatted input from a file

```
int sscanf(char *str, char *format, …)
```
Read formatted input from a string

Return value: Number of input items assigned.

Note that the arguments are pointers!

Line-Based I/O

`int puts(char *line)`

Outputs string pointed to by `line` followed by newline character to `stdout`

`char *gets(char *s)`

Reads the next input line from `stdin` into buffer pointed to by `s`

Null terminates

`char *fgets(char *s, int size, FILE * stream)`

"size" is the size of the buffer.

Stops reading before buffer overrun.

Will store the \n, if it was read.

`int getchar()`

Reads a character from stdin

Returns it as an int (0..255)

Returns EOF (i.e., -1) if "end-of-file" or "error".

General I/O

Direct system call interface

`open()` = returns an integer file descriptor

`read()`, `write()` = takes file descriptor as parameter

`close()` = closes file and file descriptor

Standard file descriptors for each process

Standard input (keyboard)

`stdin` (i.e., 0)

Standard output (display)

`stdout` (i.e., 1)

Standard error (display)

`stderr` (i.e., 2)

Error handling

Standard error (stderr)

- Used by programs to signal error conditions
- By default, stderr is sent to display
- Must redirect explicitly even if stdout sent to file

```
fprintf(stderr, "getline: error on input\n");
perror("getline: error on input");
```

- Typically used in conjunction with errno return error code
  - errno = single global variable in all C programs
  - Integer that specifies the type of error
  - Each call has its own mappings of errno to cause
  - Used with perror to signal which error occurred

Memory allocation and management

```
(void *) malloc (int numberOfBytes)
```

Dynamically allocates memory from the heap

- Memory persists between function invocations (unlike local variables)

Returns a pointer to a block of at least numberOfBytes bytes

Not zero filled!

Allocate an integer

```
int* iptr = (int*) malloc(sizeof(int));
```

Allocate a structure

```
struct name* nameptr =
   (struct name*) malloc(sizeof(struct name));
```

Allocate an integer array with "n" elements

```
int *ptr = (int *) malloc(n * sizeof(int));
```

Memory allocation and management

```
(void *) malloc (int numberOfBytes)
```

Be careful to allocate enough memory!

Overrun on the space is undefined!!!

Common error:

```
char *cp = (char *) malloc(strlen(buf)*sizeof(char))
```

NOTE: `strlen` ***doesn't account for the NULL terminator!***

Fix:

```
char *cp = (char *) malloc((strlen(buf)+1)*sizeof(char))
```

Memory allocation and management

```
void free(void * p)
```

Deallocates memory in heap.

Pass in a pointer that was returned by `malloc`.

Example

```
int* iptr = (int*) malloc(sizeof(int));
free(iptr);
```

Example

```
struct table* tp =
        (struct table*) malloc(sizeof(struct table));
free(tp);
```

Freeing the same memory block twice corrupts memory and leads to exploits!

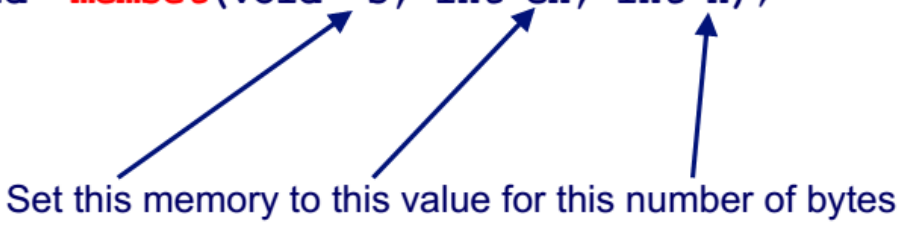
Memory allocation and management

Sometimes, before you use memory returned by malloc, you want to zero it

Or maybe set it to a specific value

`memset()` sets a chunk of memory to a specific value

```
void *memset(void *s, int ch, int n);
```



Memory allocation and management

How to move a block of bytes efficiently?

```
void *memmove(void *dest, void *src, int n);
```

How to allocate zero-filled chunk of memory?

```
void *calloc(int numberThings, int sizeOfThings);
```

Note:
These slides use "`int`"
However, "`size_t`" is better.
Makes code more portable.
"`size_t`" → unsigned integer.

Strings

String functions are provided in the string library.

```
#include <string.h>
```

Includes functions such as:

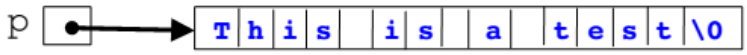
- Compute length of string
- Copy strings
- Concatenate strings
- ...

Strings

In C, a string is an array of characters terminated with the "null" character ('\0' == 0).

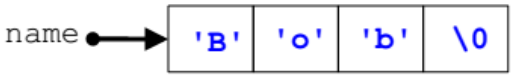
Set p to the address of a character array

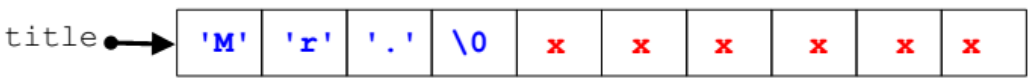
```
char *p = "This is a test";
```



NOTE: p can be reassigned to a different address

```
char name[4] = "Bob";
char title[10] = "Mr.";
```

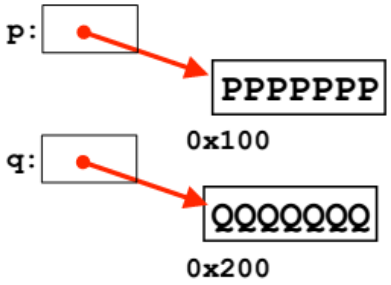




Copying strings

Consider

```
char* p ="PPPPPPP";
char* q ="QQQQQQQ";
p = q;
```



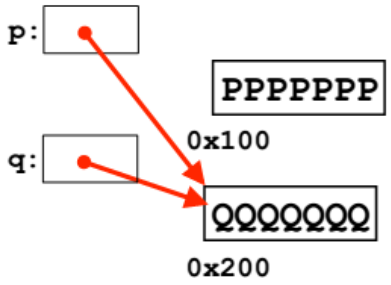
What does this do?

1. Copy QQQQQQ into 0x100?
2. Set p to 0x200

Copying strings

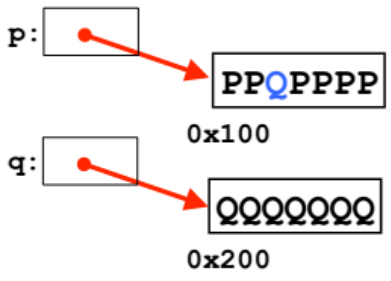
Consider

```
char* p ="PPPPPPP";
char* q ="QQQQQQQ";
p = q;
```



What does this do?

1. ~~Copy QQQQQQ into 0x100?~~
2. Set p to 0x200



To copy the strings?

- Manually copy characters

```
p[2] = q[2];
```

- Use `strncpy` to copy characters

Strings

Assignment(=) and equality (==) operators

```
char *p;
char *q;
if (p == q) {
  printf("This is only true if p and q point
                                  to the same address");
}
p = q;  /* The address contained in q is placed */
        /*    in p.  Does not change the memory */
        /*    locations p previously pointed to.*/
```

C String Library

Some of C's string functions

`strlen(char *s1)`

Returns the number of characters in the string, not including the "null" character

`strncpy(char *s1, char *s2, int n)`

Copies at most n characters of s2 on top of s1. The order of the parameters mimics the assignment operator

`strncmp (char *s1, char *s2, int n)`

Compares up to n characters of s1 with s2 lexigraphically.

Returns < 0, 0, > 0 if s1 < s2, s1 == s2 or s1 > s2

`strncat(char *s1, char *s2, int n)`

Appends at most n characters of s2 to s1

Insecure deprecated versions: `strcpy`, `strcmp`, `strcat`

Random number generation

Generate pseudo-random numbers

```
int rand(void);
```

Gets next random number

```
void srand(unsigned int seed);
```

Sets the seed for Pseudo-Random Number Generator

آشنایی با اسمبلی ARM

در این بخش ابتدا چند کد assembly را روی برد خود اجرا خواهید کرد و در انتها برخی موارد در مورد زبان assembly یادآوری می شود.

اسمبل و اجرای برنامه ی assembly :

```
pi@raspberrypi ~ $ mkdir ~/session04
pi@raspberrypi ~ $ mkdir ~/session04/asm
pi@raspberrypi ~ $ cd ~/session04/asm/
pi@raspberrypi ~/session04/asm $ nano test01.s
```

کد زیر را در فایل وارد کنید و با ctrl+o و بعد ctrl+x فایل را save و از برنامه خارج شوید. دقت کنید که در هنگامی که می خواهید فایل را save کنید، از شما پرسیده می شود که فایل با چه نامی ذخیره شود، که شما باید با وارد کردن enter، نام قبلی را تایید کنید.

```
  GNU nano 2.2.6                File: test01.s

/* -- test01.s */
/* This is a comment */
.global main /* 'main' is our entry point and must be global */
.func main   /* 'main' is a function */

main:           /* This is main */
    mov r0, #2 /* Put a 2 inside the register r0 */
    bx lr       /* Return from main */
_
```

بعد از اینکه کد را نوشته و آن را ذخیره کردید، باید توسط اسمبلر، آن را اسمبل کرده و یک object file ایجاد کنید.

```
pi@raspberrypi ~/session04/asm $ as -o ./test01.o ./test01.s_
```

سپس توسط gcc فایل object file ایجاد شده را به فایل اجرایی تبدیل می کنید.

```
pi@raspberrypi ~/session04/asm $ gcc -o test01 ./test01.o_
```

پس از اینکه فایل ایجاد شد، باید آن را اجرا کرد.توسط دستور زیر ابتدا فایل ایجاد شده را اجرا کرده و سپس مقداری را که بازگردانده باشد را می بینید.

```
pi@raspberrypi ~/session04/asm $ ./test01
pi@raspberrypi ~/session04/asm $ echo $?
2
pi@raspberrypi ~/session04/asm $
```

برنامه های زیر را نیز با نام هایی که در بالای هر عکس مشاهده می کنید، ایجاد و اسمبل و اجرا کنید.

GNU nano 2.2.6 File: /home/pi/session04/asm/load01.s

```
/* -- load01.s */
/* -- Data section */
.data
/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar1 */
myvar1:
    /* Contents of myvar1 is just 4 bytes containing value '3' */
    .word 3
/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar2 */
myvar2:
    /* Contents of myvar2 is just 4 bytes containing value '4' */
    .word 4
/* -- Code section */
.text
/* Ensure code is 4 byte aligned */
.balign 4
.global main
main:
    ldr r1, addr_of_myvar1 /* r1 ← &myvar1 */
    ldr r1, [r1]           /* r1 ← *r1 */
    ldr r2, addr_of_myvar2 /* r2 ← &myvar2 */
    ldr r2, [r2]           /* r2 ← *r2 */
    add r0, r1, r2         /* r0 ← r1 + r2 */
    bx lr
/* Labels needed to access data */
addr_of_myvar1 : .word myvar1
addr_of_myvar2 : .word myvar2
```

GNU nano 2.2.6 File: /home/pi/session04/asm/store01.s

```
/* -- store01.s */
/* -- Data section */
.data
/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar1 */
myvar1:/* Contents of myvar1 is just '3' */
    .word 0
/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar2 */
myvar2:
    /* Contents of myvar2 is just '3' */
    .word 0
/* -- Code section */
.text
/* Ensure function section starts 4 byte aligned */
.balign 4
.global main
main:
    ldr r1, addr_of_myvar1 /* r1 ← &myvar1 */
    mov r3, #3             /* r3 ← 3 */
    str r3, [r1]           /* *r1 ← r3 */
    ldr r2, addr_of_myvar2 /* r2 ← &myvar2 */
    mov r3, #4             /* r3 ← 4 */
    str r3, [r2]           /* *r2 ← r3 */
    /* Same instructions as above */
    ldr r1, addr_of_myvar1 /* r1 ← &myvar1 */
    ldr r1, [r1]           /* r1 ← *r1 */
    ldr r2, addr_of_myvar2 /* r2 ← &myvar2 */
    ldr r2, [r2]           /* r1 ← *r2 */
    add r0, r1, r2
    bx lr
/* Labels needed to access data */
addr_of_myvar1 : .word myvar1
addr_of_myvar2 : .word myvar2
```

GNU nano 2.2.6 File: branch01.s

```
/* -- branch01.s */
.text
.global main
main:
    mov r0, #2 /* r0 ← 2 */
    b end      /* branch to 'end' */
    mov r0, #3 /* r0 ← 3 */
end:
    bx lr
```

```
  GNU nano 2.2.6                File: loop01.s

/* -- loop01.s */
.text
.global main
main:
    mov r1, #0          /* r1 ← 0 */
    mov r2, #1          /* r2 ← 1 */
loop:
    cmp r2, #22         /* compare r2 and 22 */
    bgt end             /* branch if r2 > 22 to end */
    add r1, r1, r2      /* r1 ← r1 + r2 */
    add r2, r2, #1      /* r2 ← r2 + 1 */
    b loop
end:
    mov r0, r1          /* r0 ← r1 */
    bx lr
```

debug کردن کد اجرایی توسط GDB :

ابتدا باید gdb را اجرا کنید.

```
pi@raspberrypi ~/session04/asm $ gdb --args ./test01
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabihf".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/pi/session04/asm/test01...(no debugging symbols found)...done.
(gdb) _
```

پس از اینکه وارد محیط gdb شدید، با دستور start شروع میکنید.

```
(gdb) start
Temporary breakpoint 1 at 0x8390
Starting program: /home/pi/session04/asm/test01

Temporary breakpoint 1, 0x00008390 in main ()
```

توسط stepi می توانید برنامه را پیش ببرید.

```
(gdb) stepi
0x00008394 in main ()
(gdb) stepi
0x00008398 in main ()
(gdb) stepi
0x0000839c in main ()
(gdb) stepi
0x000083a0 in main ()
(gdb) _
```

توسط دستور زیر می توانید مقادیری که در رجیستر ها وجود دارد را ببینید.

```
(gdb) info registers r0 r1 r2 r3
r0             0x2 2
r1             0x7efff814 2130704404
r2             0x7efff81c 2130704412
r3             0x8390 33680
(gdb) _
```

Disassemble کردن کد اجرایی توسط GDB :

ابتدا باید gdb را اجرا کنید.

```
pi@raspberrypi ~/session04/asm $ gdb --args ./test01
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabihf".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/pi/session04/asm/test01...(no debugging symbols found)...done.
(gdb) _
```

پس از اینکه وارد محیط gdb شدید، با دستور start شروع میکنید.

```
(gdb) start
Temporary breakpoint 1 at 0x8390
Starting program: /home/pi/session04/asm/test01

Temporary breakpoint 1, 0x00008390 in main ()
```

با دستور disassemble می توانید فایل اجرایی را disassemble کنید.

```
(gdb) disassemble
Dump of assembler code for function main:
=> 0x00008390 <+0>: mov r0, #2
   0x00008394 <+4>: mov r0, #2
   0x00008398 <+8>: mov r0, #2
   0x0000839c <+12>: mov r0, #2
   0x000083a0 <+16>: bx lr
End of assembler dump.
(gdb)
```

مروری بر برنامه نویسی assembly :

شکل کلی دستورات در اسمبلی آرم به صورت زیر می باشد.

```
[<label>:]   [<instruction or directive>}   @ comment
```

GNU Assembler Directives for ARM

The follow is an alphabetical listing of the more command GNU assembler directives.

GNU Assembler Directive	Description
.ascii "<string>"	Inserts the string as data into the assembly (like DCB in armasm).
.asciz "<string>"	Like .ascii, but follows the string with a zero byte.
.balign <power_of_2> {,<fill_value> {,<max_padding>} }	Aligns the address to <power_of_2> bytes. The assembler aligns by adding bytes of value <fill_value> or a suitable default. The alignment will not occur if more than <max_padding> fill bytes are required (similar to ALIGN in armasm).
.byte <byte1> {,<byte2>} …	Inserts a list of byte values as data into the assembly (like DCB in armasm).
.code <number_of_bits>	Sets the instruction width in bits. Use 16 for Thumb and 32 for ARM assembly (similar to CODE16 and CODE32 in armasm).
.else	Use with .if and .endif (similar to ELSE in armasm).
.end	Marks the end of the assembly file (usually omitted).
.endif	Ends a conditional compilation code block – see .if, .ifdef, .ifndef (similar to ENDIF in armasm).
.endm	Ends a macro definition – see .macro (similar to MEND in armasm).
.endr	Ends a repeat loop – see .rept and .irp (similar to WEND in armasm).
.equ <symbol name>, <value>	This directive sets the value of a symbol (similar to EQU in armasm)
.err	Causes assembly to halt with an error.
.exitm	Exit a macro partway through – see .macro (similar to MEXIT in armasm)
.global <symbol>	This directive gives the symbol external linkage (similar to EXPORT in armasm).
.hword <short1> {,<short2>} …	Inserts a list of 16-bit values as data into the assembly (similar to DCW in armasm).

GNU Assembler Directive	Description
`.if <logical_expression>`	Makes a block of code conditional. End the block using .endif (similar to IF in armasm). See also .else.
`.ifdef <symbol>`	Include a block of code if <symbol> is defined. End the block with .endif.
`.ifndef <symbol>`	Include a block of code if <symbol> is not defined. End the block with .endif.
`.include "<filename>"`	Includes the indicated source file (similar to INCLUDE in armasm or #include in C).
`.irp <param> {,<val_1>} {,<val_2>} …`	Repeats a block of code, once for each value in the value list. Mark the end of the block using a .endr directive. In the repeated code block, use \<param> to substitute the associated value in the value list.

`.macro <name> {<arg_1} {,<arg_2>} … {,<arg_N>}`	Defines an assembler macro called <name> with N parameters. The macro definition must end with .endm. To escape from the macro at an earlier point, use .exitm. These directives are similar to MACRO, MEND, and MEXIT in armasm. You must precede the dummy macro parameters by \. For example:
	
	`.macro SHIFTLEFT a, b`
	`  .if \b < 0`
	`  MOV \a, \a, ASR #-\b`
	`  .exitm`
	`  .endif`
	`  MOV \a, \a, LSL #\b`
	`.endm`

`.rept <number_of_times>`	Repeats a block of code the given number of times. End with .endr.
`<register_name> .req <register_name>`	This directive names a register. It is similar to the RN directive in armasm except that you must supply a name rather than a number on the right (e.g., `acc .req r0`).
`.section <section_name> {,"<flags>"}`	Starts a new code or data section. Sections in GNU are called `.text`, a code section, `.data`, an initialized data section, and `.bss`, an uninitialized data section. These sections have default flags, and the linker understands the default names (similar directive to the armasm directive AREA). The following are allowable .section flags for ELF format files: <Flag> Meaning a allowable section w writable section x executable section
`.set <variable_name>, <variable_value>`	This directive sets the value of a variable. It is similar to SETA in armasm.
`.space <number_of_bytes> {,<fill_byte>}`	Reserves the given number of bytes. The bytes are filled with zero or <fill_byte> if specified (similar to SPACE in armasm).
`.word <word1> {,<word2>} …`	Inserts a list of 32-bit word values as data into the assembly (similar to DCD in armasm).

Assembler Special Characters / Syntax

Inline comment char:	'@'
Line comment char:	'#'
Statement separator:	';'
Immediate operand prefix:	'#' or '$'

Register Names

General registers:	%r0 - %r15	($0 = const 0)
FP registers:	%f0 - %f7	
Non-saved (temp) regs:	%r0 - %r3, %r12	
Saved registers:	%r4 - %r10	
Stack ptr register:	%sp	
Frame ptr register:	%fp	
Link (retn) register:	%lr	
Program counter:	%ip	
Status register:	$psw	
Status register flags:	xPSR	
(x = C current)	xPSR_all	
(x = S saved)	xPSR_f	
	xPSR_x	
	xPSR_ctl	
	xPSR_fs	
	xPSR_fx	
	xPSR_fc	
	xPSR_cs	
	xPSR_cf	
	xPSR_cx	
	.. and so on	

Arm Procedure Call Standard (APCS) Conventions

Argument registers:	%a0 - %a4	(aliased to %r0 - %r4)
Returned value regs:	%v1 - %v6	(aliased to %r4 - %r9)

Addressing Modes

'rn' in the following refers to any of the numbered registers, but not the control registers.

addr	Absolute addressing mode
%rn	Register direct
[%rn]	Register indirect or indexed
[%rn,#n]	Register based with offset
#imm	Immediate data

Data Sizes and Instruction Set

- The ARM is a 32-bit architecture.
- When used in relation to the ARM:
  - **Byte** means 8 bits
  - **Halfword** means 16 bits (two bytes)
  - **Word** means 32 bits (four bytes)
- Most ARM's implement two instruction sets
  - 32-bit ARM Instruction Set
  - 16-bit Thumb Instruction Set
- Jazelle cores can also execute Java bytecode

Processor Modes

- The ARM has seven basic operating modes:
  - **User** : unprivileged mode under which most tasks run
  - **FIQ** : entered when a high priority (fast) interrupt is raised
  - **IRQ** : entered when a low priority (normal) interrupt is raised
  - **Supervisor** : entered on reset and when a Software Interrupt instruction is executed
  - **Abort** : used to handle memory access violations
  - **Undef** : used to handle undefined instructions
  - **System** : privileged mode using the same registers as user mode

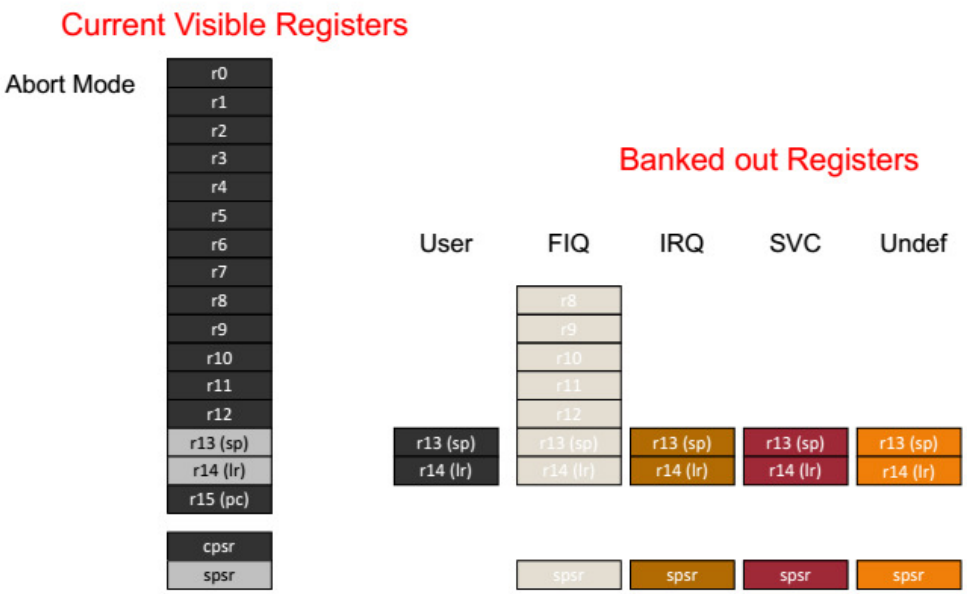
ARM Registers

- 16 Registers, 32Bits each
- R0 – R12
  - General purpose registers
- R13
  - Stack Pointer
- R14
  - Subroutine Link Register (LR)
  - Stores return address of subroutine
  - R14 stores a copy of R15 when BL instruction (Branch with Link) occurs
- R15
  - Program Counter
  - R15[1:0] always zero in ARM state
  - R15[0] always zero in Thumb State

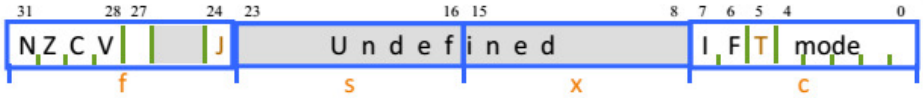
ARM Registers

- CPSR
  - Current Program Status Register
  - Contains condition code flags
- SPSR
  - Saved Program Status Register
  - A copy of CPSR

ARM Register Set



CPSR Register



- Condition code flags
  - N = Negative result from ALU
  - Z = Zero result from ALU
  - C = ALU operation Carried out
  - V = ALU operation oVerflowed
- J bit
  - Architecture 5TEJ only
  - J = 1: Processor in Jazelle state
- Interrupt Disable bits.
  - I = 1: Disables the IRQ.
  - F = 1: Disables the FIQ.
- T Bit
  - Architecture xT only
  - T = 0: Processor in ARM state
  - T = 1: Processor in Thumb state
- Mode bits
  - Specify the processor mode

Features of ARM instruction set

- Load-store architecture
- 3-address instructions
- Conditional execution of every instruction
- Possible to load/store multiple register at once
- Possible to combine shift and ALU operations in a single instruction

Instruction set

```
MOV<cc><S>  Rd, <operands>

MOVCS R0, R1 @ if carry is set
             @ then R0:=R1

MOVS  R0, #0 @ R0:=0
             @ Z=1, N=0
             @ C, V unaffected
```

Instruction set

- Data processing (Arithmetic and Logical)
- Data movement
- Flow control

Arithmetic

```
ADD   R0, R1, R2        @ R0 = R1+R2
ADC   R0, R1, R2        @ R0 = R1+R2+C
SUB   R0, R1, R2        @ R0 = R1-R2
SBC   R0, R1, R2        @ R0 = R1-R2+C-1
RSB   R0, R1, R2        @ R0 = R2-R1
RSC   R0, R1, R2        @ R0 = R2-R1+C-1
```

Bitwise logic

```
• AND   R0, R1, R2        @ R0 = R1 and R2
• ORR   R0, R1, R2        @ R0 = R1 or  R2
• EOR   R0, R1, R2        @ R0 = R1 xor R2
• BIC   R0, R1, R2        @ R0 = R1 and (~R2)
```

bit clear: R2 is a mask identifying which bits of R1 will be cleared to zero

```
R1=0x11111111     R2=0x01100101

BIC R0, R1, R2

R0=0x10011010
```

Register movement

```
• MOV   R0, R2          @ R0 = R2
• MVN   R0, R2          @ R0 = ~R2
```

move negated

Comparison

- These instructions do not generate a result, but set condition code bits (N, Z, C, V) in CPSR. Often, a branch operation follows to change the program flow.

- `CMP  R1, R2  @ set cc on R1-R2`
  compare
- `CMN  R1, R2  @ set cc on R1+R2`
  compare negated
- `TST  R1, R2  @ set cc on R1 and R2`
  bit test
- `TEQ  R1, R2  @ set cc on R1 xor R2`
  test equal

Addressing modes

- Register operands

```
ADD  R0, R1, R2
```

- Immediate operands

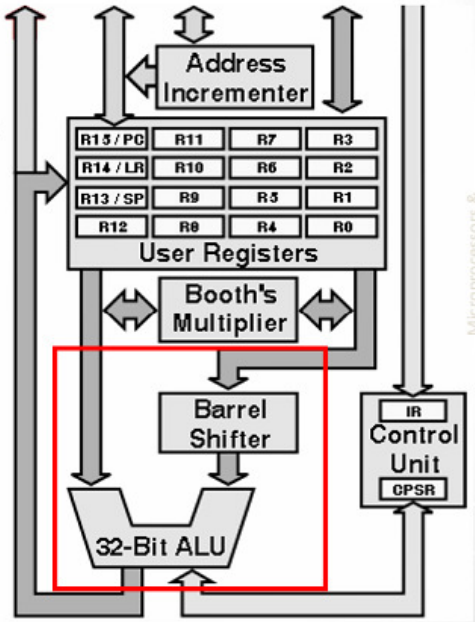
a literal;

```
ADD  R3, R3, #1     @ R3:=R3+1
AND  R8, R7, #0xff  @ R8=R7[7:0]
```

a hexadecimal literal
This is assembler dependent syntax.

Shifted register operands

- One operand to ALU is routed through the Barrel shifter. Thus, the operand can be modified before it is used. Useful for dealing with lists, table and other complex data structure. (similar to the displacement addressing mode in CISC.)



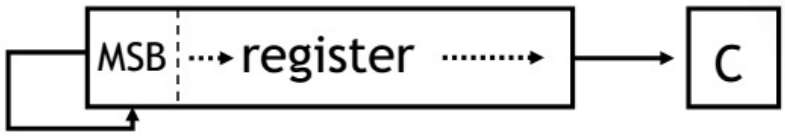
Logical shift left


```
MOV  R0, R2, LSL #2 @ R0:=R2<<2
                    @ R2 unchanged
Example: 0…0 0011 0000
Before R2=0x00000030
After  R0=0x000000C0
       R2=0x00000030
```

Logical shift right

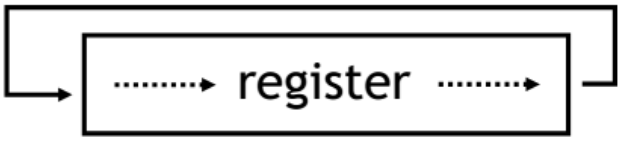

```
MOV  R0, R2, LSR #2 @ R0:=R2>>2
                    @ R2 unchanged
Example: 0…0 0011 0000
Before R2=0x00000030
After  R0=0x0000000C
       R2=0x00000030
```

Arithmetic shift right




```
MOV  R0, R2, ASR #2 @ R0:=R2>>2
                    @ R2 unchanged
Example: 1010 0…0 0011 0000
Before R2=0xA0000030
After  R0=0xE800000C
       R2=0xA0000030
```

Rotate right




```
MOV  R0, R2, ROR #2 @ R0:=R2 rotate
                    @ R2 unchanged
Example: 0…0 0011 0001
Before R2=0x00000031
After  R0=0x4000000C
       R2=0x00000031
```

Rotate right extended


```
MOV  R0, R2, RRX    @ R0:=R2 rotate
                    @ R2 unchanged
Example: 0…0 0011 0001
Before R2=0x00000031, C=1
After  R0=0x80000018, C=1
       R2=0x00000031
```

Shifted register operands

- It is possible to use a register to specify the number of bits to be shifted; only the bottom 8 bits of the register are significant.

```
ADD   R0, R1, R2, LSL R3   @
R0:=R1+R2*2^R3
```

Setting the condition codes

- Any data processing instruction can set the condition codes if the programmers wish it to

64-bit addition

```
ADDS   R2, R2, R0
ADC    R3, R3, R1
```

	R1	R0
+	R3	R2
	R3	R2

Multiplication

- `MUL R0, R1, R2 @ R0 = (R1xR2)[31:0]`
- Features:
  - Second operand can't be immediate
  - The result register must be different from the first operand
  - If S bit is set, C flag is meaningless
- See the reference manual (4.1.33)

Multiplication

- Multiply-accumulate

```
MLA  R4, R3, R2, R1  @ R4 = R3xR2+R1
```

- Multiply with a constant can often be more efficiently implemented using shifted register operand

```
MOV  R1, #35
MUL  R2, R0, R1
   or
ADD  R0, R0, R0, LSL #2  @ R0'=5xR0
RSB  R2, R0, R0, LSL #3  @ R2 =7xR0'
```

Data transfer instructions

- Move data between registers and memory
- Three basic forms
  - Single register load/store
  - Multiple register load/store
  - Single register swap: **SWP(B)**, atomic instruction for semaphore

Single register load/store

- The data items can be a 8-bitbyte, 16-bit half-word or 32-bit word.

```
LDR   R0, [R1]    @ R0 := mem32[R1]
STR   R0, [R1]    @ mem32[R1] := R0
```

LDR, LDRH, LDRB for 32, 16, 8 bits

STR, STRH, STRB for 32, 16, 8 bits

Load an address into a register

- The pseudo instruction **ADR** loads a register with an address

```
table:    .word     10
…
          ADR   R0, table
```

- Assembler transfer pseudo instruction into a sequence of appropriate instructions

```
sub       r0, pc, #12
```

Addressing modes

- Memory is addressed by a register and an offset.

```
LDR   R0, [R1] @ mem[R1]
```

- Three ways to specify offsets:
  - Constant

```
LDR   R0, [R1, #4]  @ mem[R1+4]
```

  - Register

```
LDR   R0, [R1, R2]    @ mem[R1+R2]
```

  - Scaled

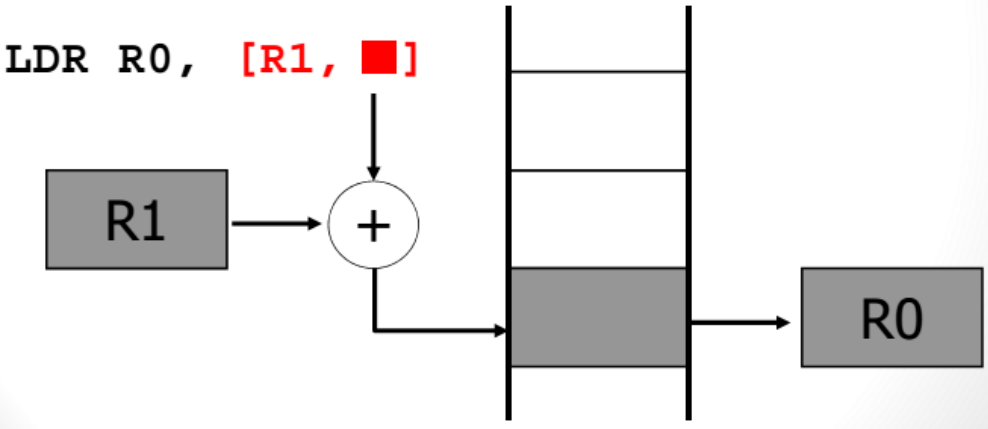
```
                      @ mem[R1+4*R2]
LDR   R0, [R1, R2, LSL #2]
```

Addressing modes

- Pre-indexed addressing (`LDR R0, [R1, #4]`) without a writeback
- Auto-indexing addressing (`LDR R0, [R1, #4]!`) calculation before accessing with a writeback
- Post-indexed addressing (`LDR R0, [R1], #4`) calculation after accessing with a writeback

Pre-indexed addressing

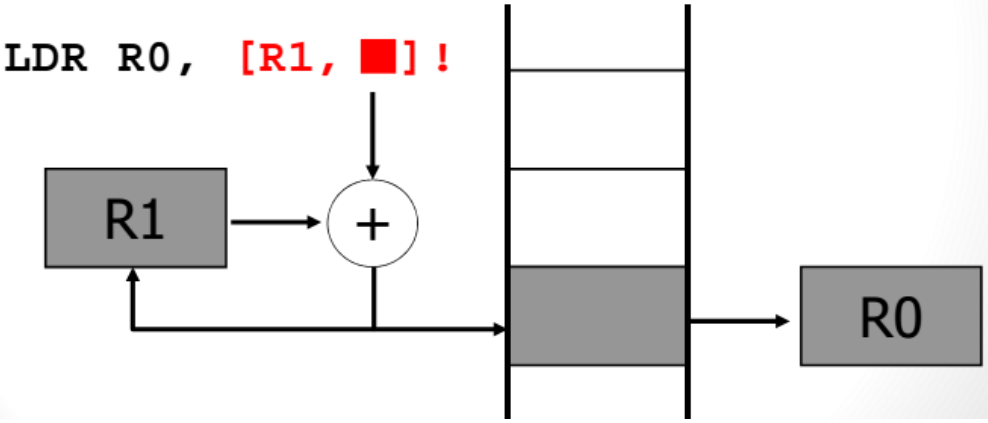
```
LDR  R0, [R1, #4]    @ R0=mem[R1+4]
                     @ R1 unchanged
```



Auto-indexing addressing

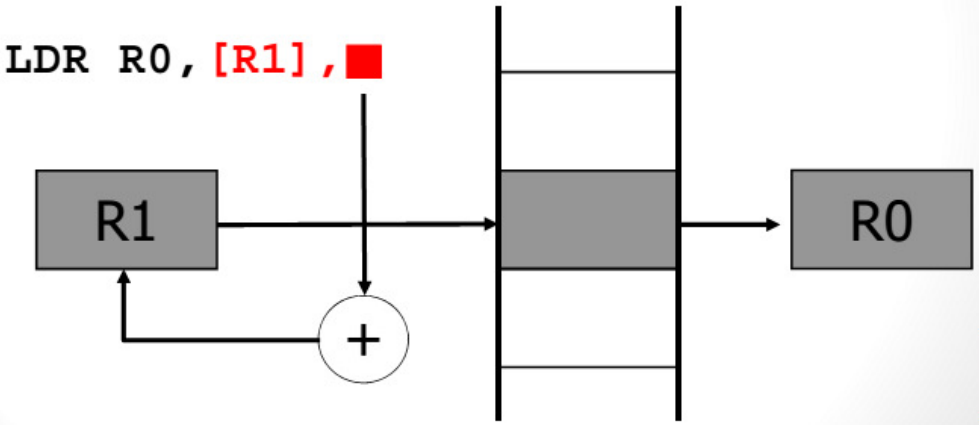
```
LDR   R0, [R1, #4]!   @ R0=mem[R1+4]
                      @ R1=R1+4
```

No extra time; Fast;



Post-indexed addressing

```
LDR   R0, R1, #4      @ R0=mem[R1]
                      @ R1=R1+4
```



Comparisons

- Pre-indexed addressing

```
LDR   R0, [R1, R2]   @ R0=mem[R1+R2]
                     @ R1 unchanged
```

- Auto-indexing addressing

```
LDR   R0, [R1, R2]!  @ R0=mem[R1+R2]
                     @ R1=R1+R2
```

- Post-indexed addressing

```
LDR   R0, [R1], R2   @ R0=mem[R1]
                     @ R1=R1+R2
```

Control flow instructions

- Determine the instruction to be executed next
- Branch instruction

```
          B   label
          …
label:    …
```

- Conditional branches

```
          MOV   R0, #0
loop:          …
          ADD   R0, R0, #1
          CMP   R0, #10
          BNE   loop
```

Branch conditions

Branch	Interpretation	Normal uses
B BAL	Unconditional Always	Always take this branch Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC BLO	Carry clear Lower	Arithmetic operation did not give carry-out Unsigned comparison gave lower
BCS BHS	Carry set Higher or same	Arithmetic operation gave carry-out Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occured
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

Branch and link

- **BL** instruction save the return address to **R14** (lr)

```
    BL    sub      @ call sub
    CMP   R1, #5   @ return to here
    MOVEQ R1, #0
    …
sub:…              @ sub entry point
    …
    MOV   PC, LR   @ return
```

Conditional execution

- Almost all ARM instructions have a condition field which allows it to be executed conditionally.

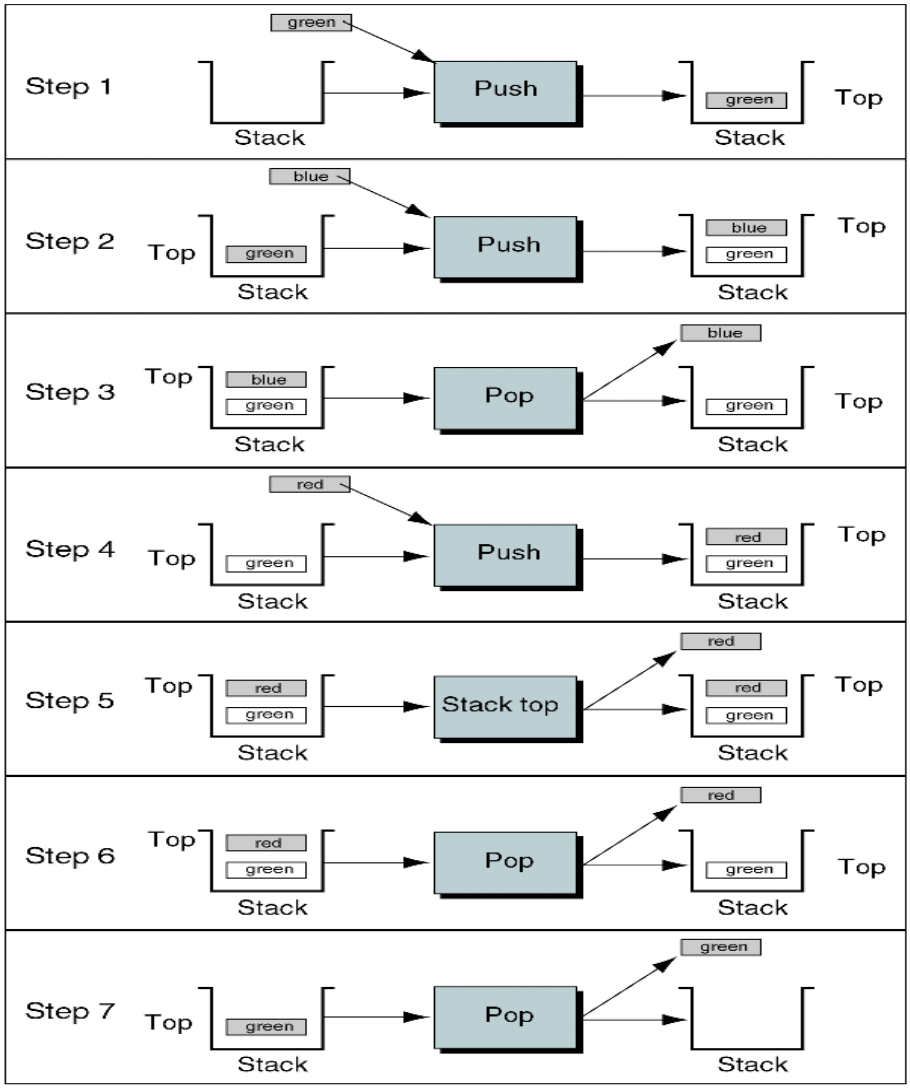
```
movcs R0, R1
```

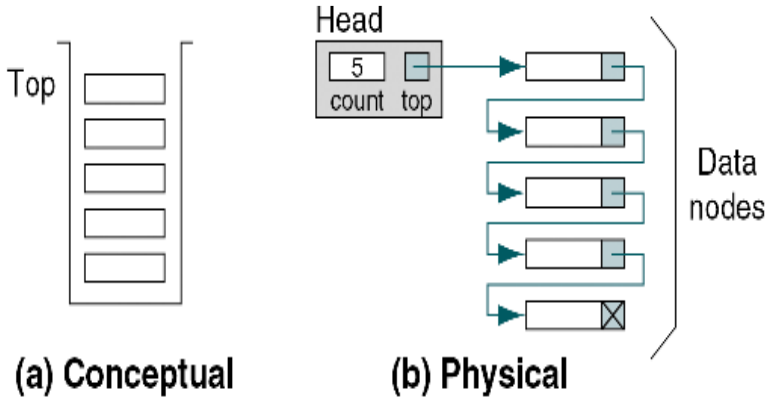
Mnemonic	Condition	Mnemonic	Condition
CS	*C*arry *S*et	CC	*C*arry *C*lear
EQ	*E*qual (Zero Set)	NE	*N*ot *E*qual (Zero Clear)
VS	O*v*erflow *S*et	VC	O*v*erflow *C*lear
GT	*G*reater *T*han	LT	*L*ess *T*han
GE	*G*reater Than or *E*qual	LE	*L*ess Than or *E*qual
PL	*Pl*us (Positive)	MI	*Mi*nus (Negative)
HI	*Hi*gher Than	LO	*Lo*wer Than (aka CC)
HS	*H*igher or *S*ame (aka CS)	LS	*L*ower or *S*ame

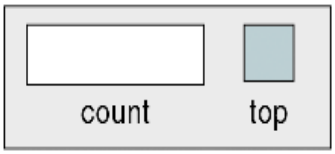
Instruction set

Operation Mnemonic	Meaning	Operation Mnemonic	Meaning
ADC	Add with Carry	MVN	Logical NOT
ADD	Add	ORR	Logical OR
AND	Logical AND	RSB	Reverse Subtract
BAL	Unconditional Branch	RSC	Reverse Subtract with Carry
B⟨*cc*⟩	Branch on Condition	SBC	Subtract with Carry
BIC	Bit Clear	SMLAL	Mult Accum Signed Long
BLAL	Unconditional Branch and Link	SMULL	Multiply Signed Long
BL⟨*cc*⟩	Conditional Branch and Link	STM	Store Multiple
CMP	Compare	STR	Store Register (Word)
EOR	Exclusive OR	STRB	Store Register (Byte)
LDM	Load Multiple	SUB	Subtract
LDR	Load Register (Word)	SWI	Software Interrupt
LDRB	Load Register (Byte)	SWP	Swap Word Value
MLA	Multiply Accumulate	SWPB	Swap Byte Value
MOV	Move	TEQ	Test Equivalence
MRS	Load SPSR or CPSR	TST	Test
MSR	Store to SPSR or CPSR	UMLAL	Mult Accum Unsigned Long
MUL	Multiply	UMULL	Multiply Unsigned Long

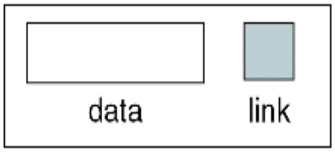
راهنمایی در مورد نحوه ی پیاده سازی stack به زبان C







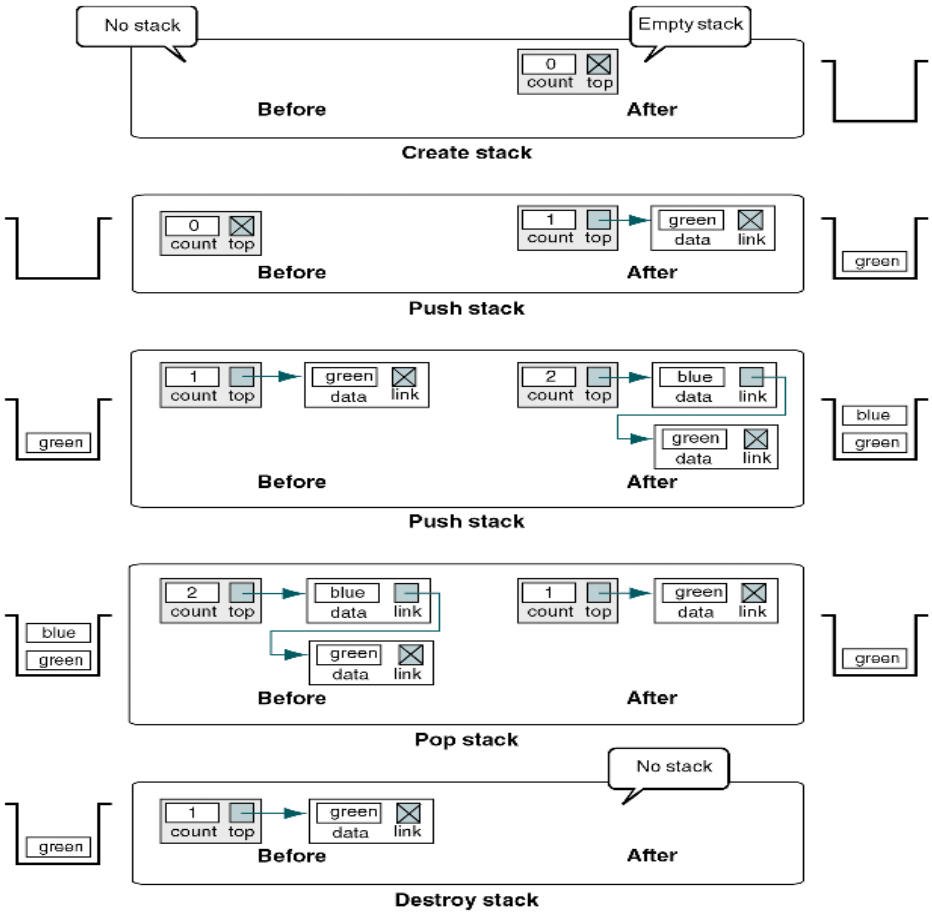
Stack head structure



Stack node structure

```
stack
  count
  top
end stack


node
  data
  link
end node
```


No stack
Empty stack
0
count top
Before
After
Create stack
0
count top
1
count top
green
data link
green
Before
After
Push stack
1
count top
green
data link
2
count top
blue
data link
green
data link
blue
green
Before
After
Push stack
2
count top
blue
data link
green
data link
1
count top
green
data link
blue
green
green
Before
After
Pop stack
No stack
1
count top
green
data link
green
Before
After
Destroy stack

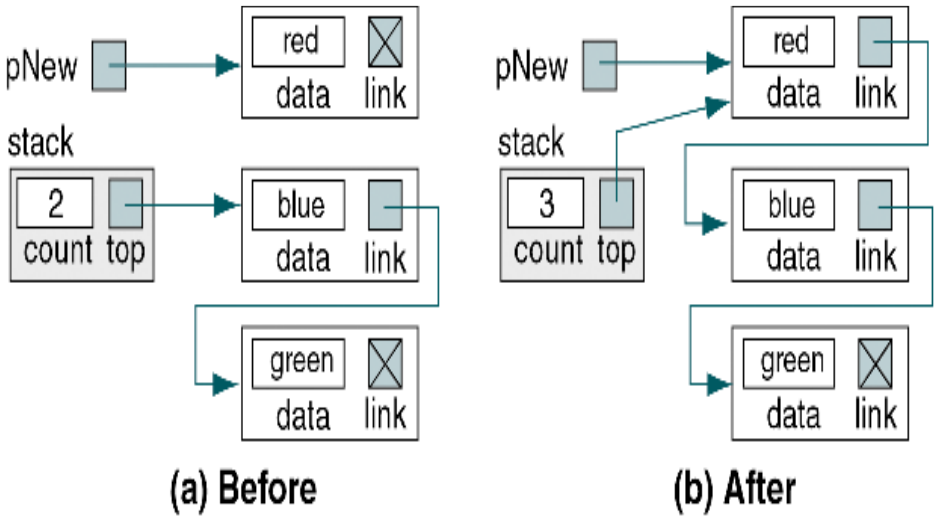


FIGURE 3-9 Push Stack Example

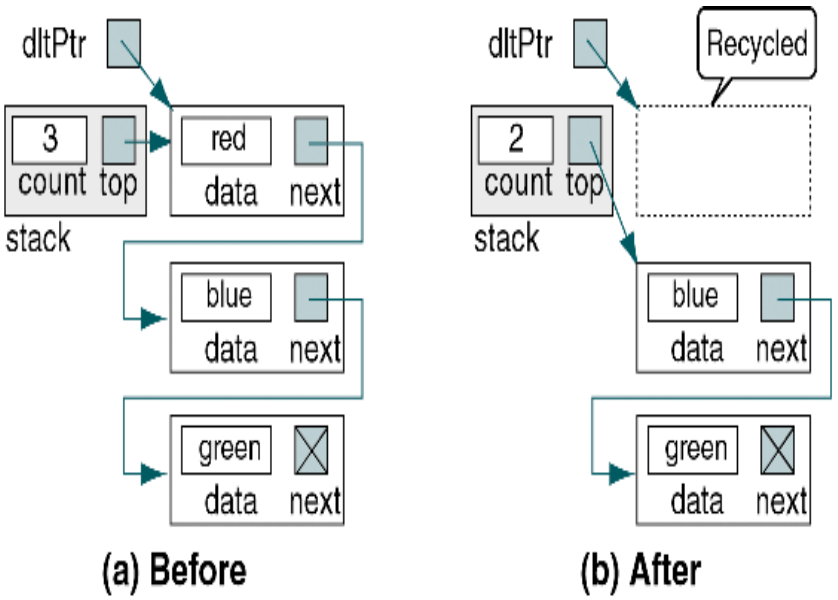


FIGURE 3-10 Pop Stack Example

```c
/*
 * C Program to Implement Stack Operations using Dynamic Memory
 * Allocation
 */
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int data;
    struct node *link;
}*top = NULL;

#define MAX 5

// function prototypes
void push();
void pop();
void empty();
void stack_full();
void stack_count();
void destroy();
void print_top();

void main()
{
    int choice;

    while (1)
    {
        printf("1. push an element \n");
        printf("2. pop an element \n");
        printf("3. check if stack is empty \n");
        printf("4. check if stack is full \n");
        printf("5. count/display elements present in stack \n");
        printf("6. empty and destroy stack \n");
        printf("7. Print top of the stack \n");
        printf("8. exit \n");
        printf("Enter your choice \n");
        scanf("%d",&choice);
        switch (choice)
        {
        case 1:
            push();
            break;
        case 2:
            pop();
            break;
        case 3:
```

```
            empty();
            break;
        case 4:
            stack_full();
            break;
        case 5:
            stack_count();
            break;
        case 6:
            destroy();
            break;
        case 7:
            print_top();
            break;
        case 8:
            exit(0);
        default:
            printf("wrong choice\n");
        }
    }
}

// to insert elements in stack
void push()
{
...
}

// to delete elements from stack
void pop()
{
...
}

// to check if stack is empty
void empty()
{
...
}

// to check if stack is full
void stack_full()
{
...
}

// to count the number of elements
void stack_count()
{
    ...
}
```

```
int st_count()
{
    ...
}

// to empty and destroy the stack
void destroy()
{
...
}

// to print top element of stack
void print_top()
{
...
}
```